Formal Approaches to Ensuring the Safety of Space Software

Ewen Denney & Bernd Fischer
{**edenney|fisch**}**@email.arc.nasa.gov**
Robust Software Engineering Group
USRA/RIACS, NASA Ames Research Center

Joint work with J. Schumann, M. Whalen, and many others

A peek into the future…

"All the News That's Fit to Print"

The New York Times

Late Edition

NEW YORK, WEDNESDAY, SEPTEMBER 19 2012

NASA FINDS FIRST CLUES IN MARS LANDING DISASTER; AUTOMATIC CODE GENERATOR FAILED, SOURCES SAY

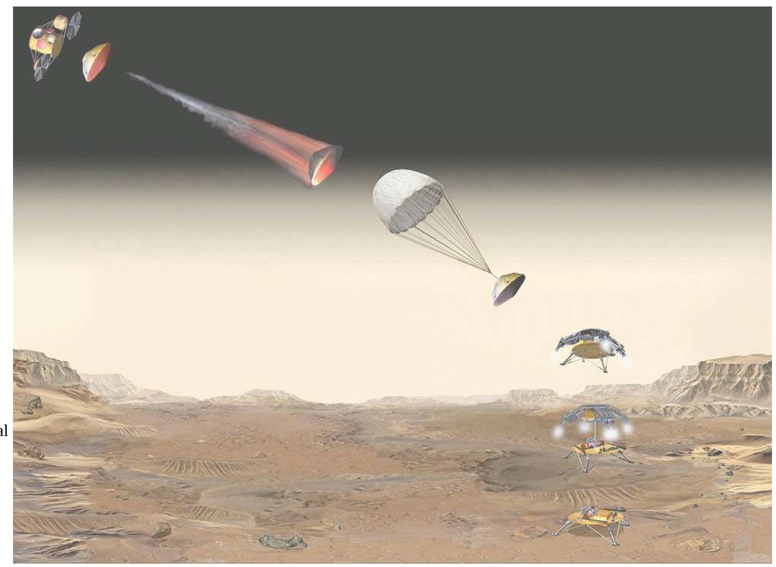
NASA Dismissed Researchers Who Investigated its Safety

By David Alfano

Mountain View, California −− NASA might have involuntarily contributed to yesterday's failed Mars landing by cutting back on its research efforts on automated code generation techniques. "Up to about 2005 we had a very strong group at the agency's Ames Research Center. But then NASA headquarters cut funding and the researchers werre hired by universities all over the place," says Dr. Michael R. Lowry, who led the efforts until 2007 and is now the Senior Software Research advisor for the entire agency. Dr. Eoghan Denney, now at the South Pacific Research Institute for New Technologies in Funafuti, Tuvalu, con− firms this. "At some point, all we did was writing proposals, proposals, proposals... Our research suffered and it was very little fun, and eventually everybody left for nicer places." Other scientists have echoed the same complaints, as sources in different funding agencies report.

A spokesperson for BreezeBrook Inc., the company that developed the code generator that is now under suspicion, also voiced similar concerns. "We feel that it is NASA's own fault. The manual clearly states that we give no warranty and that the generated code is not fit for critical use. They've cut corners and got bitten badly, but it is their own fault, really."

Legal experts think that a protracted court battle for the $3.2bn loss will fol− low, independent of the Senate's expert *(continued on page A4)*



Mars landing scenario; software experts at Ames assume that the landing gear was not deployed properly

Manned Mission On Hold for Now

By Kevin H. Knuth

Houston, Texas −− NASA officials confirmed that the manned mission planned for later this year is on hold for now but all four astronauts remain in training, including talk radio host Howard Stern, who paid $1.7bn for a spot.

Mars−Astronaut Howard Stern in Training

SENATE COMMISIONS EXPERT STUDY

A peek into the future…

The New York Times

"All the News That's Fit to Print"

Late Edition

NEW YORK, WEDNESDAY, SEPTEMBER 19 2012

NASA FINDS FIRST CLUES IN MARS LANDING DISASTER; AUTOMATIC CODE GENERATOR FAILED, SOURCES SAY

How do you make sure your code generator is safe?

NASA Dismissed Researchers Who Investigated its Safety

By David Alfano

Mountain View, California −− NASA might have involuntarily contributed to yesterday's failed Mars landing by cutting back on its research efforts in automated code generation techniques. "Up to about 2005 we had a very strong group at the agency's Ames Research Center. But then NASA headquarters cut funding and the researchers were hired by universities all over the place," says Dr. Michael R. Lowry, who led the efforts until 2007 and is now the Senior Software Research advisor for the entire agency. Dr. Eoghan Denney, now at the South Pacific Research Institute for New Technologies in Funafuti, Tuvalu, confirms this. "At some point, all we did was writing proposals, proposals, proposals... Our research suffered and it was very little fun, and eventually everybody left for nicer places." Other scientists have echoed the same complaints, as sources in different funding agencies report.

A spokesperson for BreezeBook Inc., the company that developed the code generator that is now under suspicion, also voiced similar concerns. "We feel that it is NASA's own fault. The manual clearly states that we give no warranty and that the generated code is not fit for critical use. They've cut corners and got bitten badly, but it is their own fault, really."

Legal experts think that a protracted court battle for the $3.2bn loss will follow, independent of the Senate's expert

(continued on page A4)

Mars landing scenario; software experts at Ames assume that the landing gear was not deployed properly

Manned Mission On Hold for Now

By Kevin H. Knuth

Houston, Texas −− NASA officials confirmed that the manned mission planned for later this year is on hold for now but all four astronauts remain in training, including talk radio host Howard Stern, who paid $1.7bn for a spot.

Mars−Astronaut Howard Stern in Training

SENATE COMMISIONS EXPERT STUDY

Outline

1. Introduction (*or: Taking Stock*)
2. Certifiable Program Generation (*or: I have a plan*)
3. Certification Framework (*or: Greek Letters*)
4. Annotation Generation (*or: TANSTAAFL*)
5. Experiments (*or: Drosophila and Tables*)
6. Future Work (*or: Wild Speculations*)

Taking Stock: The Good, the Bad, the Ugly

The Good: It hasn’t happened yet!

- no accidents caused by generated code

Taking Stock: The Good, the Bad, the Ugly

The Good: It hasn’t happened yet!

- no accidents caused by generated code

The Bad: It hasn’t happened yet!

- limited generator capabilities: glorified pretty-printers
- limited generator usage
- excessive post-hoc validation

Taking Stock: The Good, the Bad, the Ugly

The Good: It hasn't happened yet!

- no accidents caused by generated code

The Bad: It hasn't happened yet!

- limited generator capabilities: glorified pretty-printers
- limited generator usage
- excessive post-hoc validation

The Ugly: It will happen!

- too many bug reports (cf. optimizing compilers):
  Notice the function "beforeStart" should return a boolean but doesn't. Since this code was generated by Netbeans it's not editable…
- too many generators (`www.codegeneration.net`: $\approx$ 200)
- increasing application pressure: model-driven architecture

Taking Stock: The Correctness Dilemma

Do you trust your code generator?

- Correctness of generated code depends on correctness of generator
- Correctness of generator difficult to show practically
  - very large
  - very complicated
  - very dynamic

Spec

```
const nat n := 6 as 'Number of state variables'.
data double f(1..3, time) as 'gyro readings'.
double x(1..n) as 'state variable vector'.
double u(1..n) as 'process noise vector'.
double q(1..n) as 'variance of process noise'.
u(I) ~ gauss(0, q(I)).
equations process_eqs are [
  dot x(1) := (hat x(4) - x(4)) - u(1)
              + x(2) * (f(3)@t - hat x(6))
              - x(3) * (f(2)@t - hat x(5)),
  ...
  dot x(6) := u(6)
].
...
```

$\Rightarrow$

$\Rightarrow$

Code

```
// Calculate KH
for(i = 0; i <= 5; i++)
  for(j = 0; j <= 5; j++)
    tmp0 = 0;
    for(k = 0; k <= 2; k++)
      tmp0 += gain[i][k] * h[k][j];
    tmp1[i][j] = tmp0;
// Calculate I-KH
for(i = 0; i <= 5; i++)
  for(j = 0; j <= 5; j++)
    tmp2[i][j] = id[i][j] - tmp1[i][j];
...
```

So what to do?

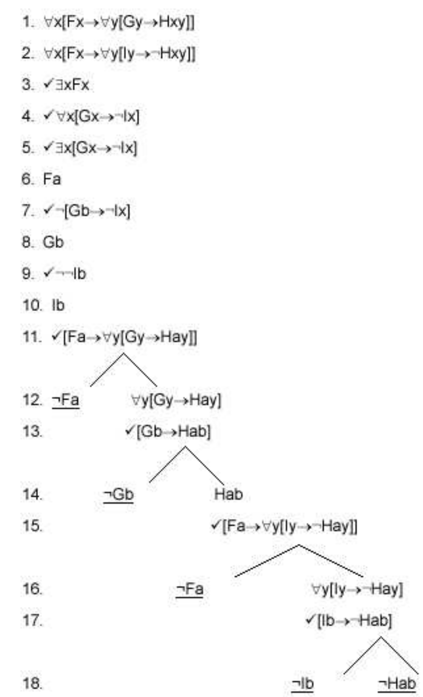
Generator Assurance Approaches (I)

Correctness-by-Construction:

Generator is based on logical framework; code is derived by correctness-preserving transformations

Techniques:

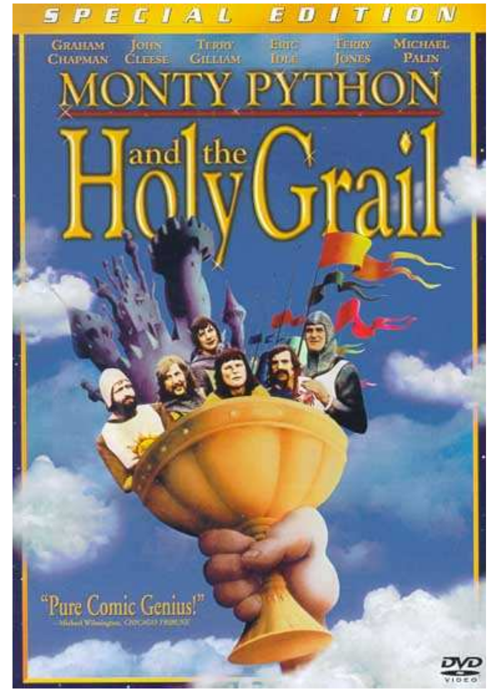
- deductive program synthesis
- refinement and transformation systems
- translation verification



Generator Assurance Approaches (I)

Correctness-by-Construction:

Generator is based on logical framework; code is derived by correctness-preserving transformations



Techniques:

- deductive program synthesis
- refinement and transformation systems
- translation verification

Advantages:

- highest degree of confidence ("proofs-as-programs")

Disadvantages:

- expensive — systems difficult to build & maintain
- opaque — correctness argument convoluted and buried in generator

  ($\Rightarrow$ must trust generator)

Generator Assurance Approaches (II)

Generator Qualification:

Generator is tested to same level of criticality as generated code

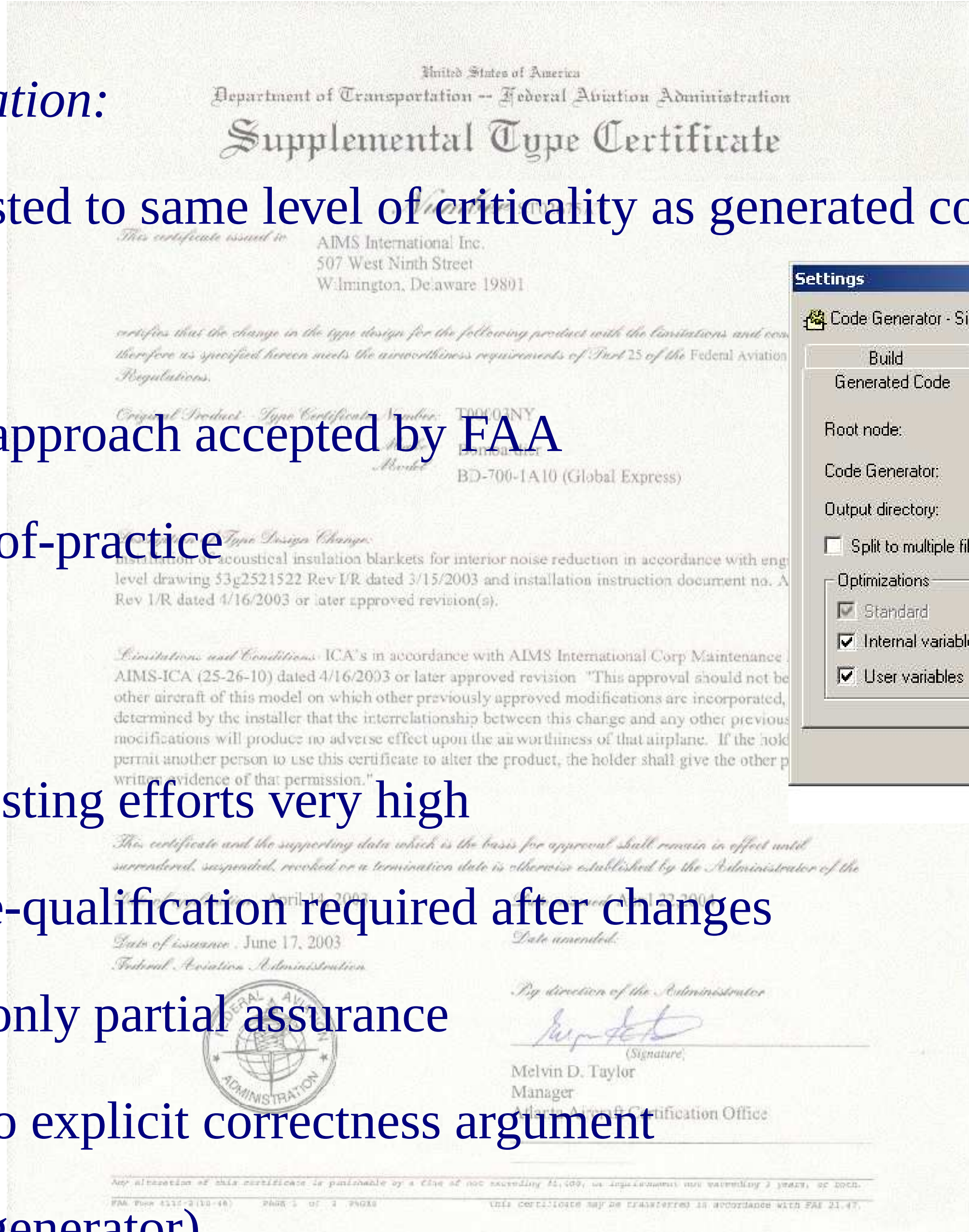
Advantages:

- currently only approach accepted by FAA
- currently state-of-practice

Disdvantages:

- expensive — testing efforts very high
- expensive — re-qualification required after changes
- limited — only partial assurance
- opaque — no explicit correctness argument

(⇒ must trust generator)


United States of America
Department of Transportation -- Federal Aviation Administration
Supplemental Type Certificate

This certificate issued to AIMS International Inc.
507 West Ninth Street
Wilmington, Delaware 19801

certifies that the change in the type design for the following product with the limitations and conditions therefore as specified hereon meets the airworthiness requirements of Part 25 of the Federal Aviation Regulations.

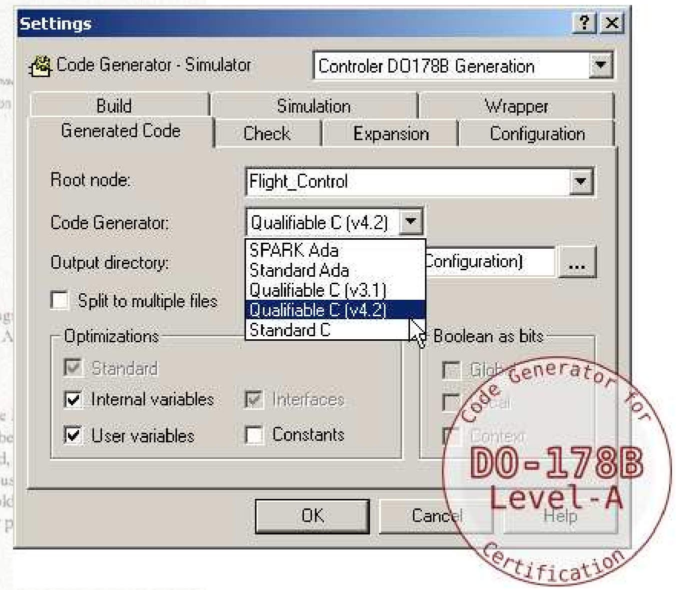
Original Product - Type Certificate Number: T00003NY
Model: BD-700-1A10 (Global Express)

Description of Type Design Change: Installation of acoustical insulation blankets for interior noise reduction in accordance with engineering level drawing 53g2521522 Rev I/R dated 3/15/2003 and installation instruction document no. A... Rev 1/R dated 4/16/2003 or later approved revision(s).

Limitations and Conditions: ICA's in accordance with AIMS International Corp Maintenance ... AIMS-ICA (25-26-10) dated 4/16/2003 or later approved revision. "This approval should not be ... other aircraft of this model on which other previously approved modifications are incorporated, ... determined by the installer that the interrelationship between this change and any other previous modifications will produce no adverse effect upon the airworthiness of that airplane. If the hold... permit another person to use this certificate to alter the product, the holder shall give the other p... written evidence of that permission."

This certificate and the supporting data which is the basis for approval shall remain in effect until surrendered, suspended, revoked or a termination date is otherwise established by the Administrator of the ...

Date of issuance: June 17, 2003
Date amended:
Federal Aviation Administration
By direction of the Administrator
(Signature)
Melvin D. Taylor
Manager
Atlanta Aircraft Certification Office



Taking Stock: The Correctness Dilemma (revisited)

Do you trust your code generator?

- Correctness of generated code depends on correctness of generator
- Correctness of generator difficult to show practically
  - very large
  - very complicated
  - very dynamic

Spec

```
const nat n := 6 as 'Number of state variables'.
data double f(1..3, time) as 'gyro readings'.
double x(1..n) as 'state variable vector'.
double u(1..n) as 'process noise vector'.
double q(1..n) as 'variance of process noise'.
u(I) ~ gauss(0, q(I)).
equations process_eqs are [
  dot x(1) := (hat x(4) - x(4)) - u(1)
              + x(2) * (f(3)@t - hat x(6))
              - x(3) * (f(2)@t - hat x(5)),
  ...
  dot x(6) := u(6)
].
...
```

$\Rightarrow$

$\Rightarrow$

Code

```
// Calculate KH
for(i = 0; i <= 5; i++)
  for(j = 0; j <= 5; j++)
    tmp0 = 0;
    for(k = 0; k <= 2; k++)
      tmp0 += gain[i][k] * h[k][j];
    tmp1[i][j] = tmp0;
// Calculate I-KH
for(i = 0; i <= 5; i++)
  for(j = 0; j <= 5; j++)
    tmp2[i][j] = id[i][j] - tmp1[i][j];
...
```

So what?

- Don't care whether generator is buggy for other people
  as long as it works for me now!

$\Rightarrow$ Certifiable Program Generation

Certifiable Program Generation

Basic Idea I:

Certify generated programs individually, not the generator

$\Rightarrow$ product-oriented approach rather than process-oriented

$\Rightarrow$ no need to re-certify generator

$\Rightarrow$ minimizes trusted component base

Certifiable Program Generation

Basic Idea I:

Certify generated programs individually, not the generator

Basic Idea II:

Extend the generator to support certification

⇒ generate code with additional “mark-up”

⇒ CAVEAT: keep certification independent from code generation

Certifiable Program Generation

Basic Idea I:

Certify generated programs individually, not the generator

Basic Idea II:

Extend the generator to support certification

Basic Idea III:

Use Floyd-Hoare program verification techniques

$\Rightarrow$ rigorous mathematical foundation

$\Rightarrow$ proofs are independently verifiable evidence (*certificates*)

$\Rightarrow$ code mark-up gives hints only

$\Rightarrow$ code mark-up $\hat{=}$ pre-/post-conditions, loop invariants

Certifiable Program Generation

Basic Idea I:

Certify generated programs individually, not the generator

Basic Idea II:

Extend the generator to support certification

Basic Idea III:

Use Floyd-Hoare program verification techniques

Basic Idea IV:

Focus on specific *safety* properties

- array bounds, partial operators, ...
- variable initialization, def-use, ...

language-specific

- physical units, frames, ...
- volatile memory restrictions, ...
- vector norms, matrix symmetry, ...

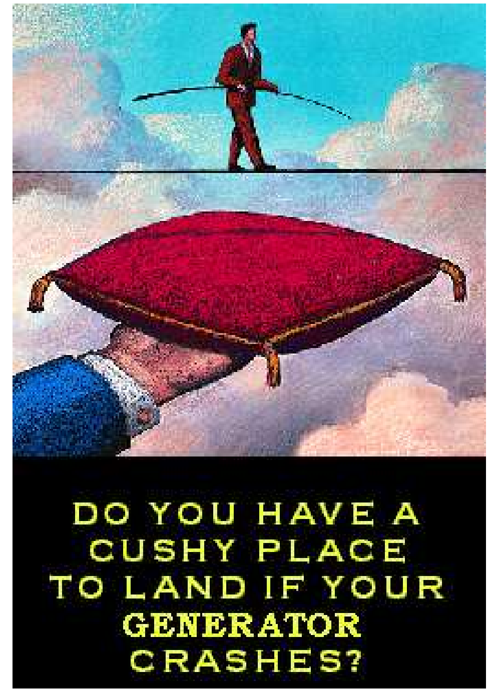
domain-specific

- ...

Generator Assurance Approaches (III)

Certifiable Program Generation:

Generator is extended to generate code with extra artefacts that support an independent assurance demonstration



Related techniques:

- result checking
- proof-carrying code

Advantages:

- customizable — different safety properties
- transparent — explicit safety arguments
- high degree of assurance — formal proofs

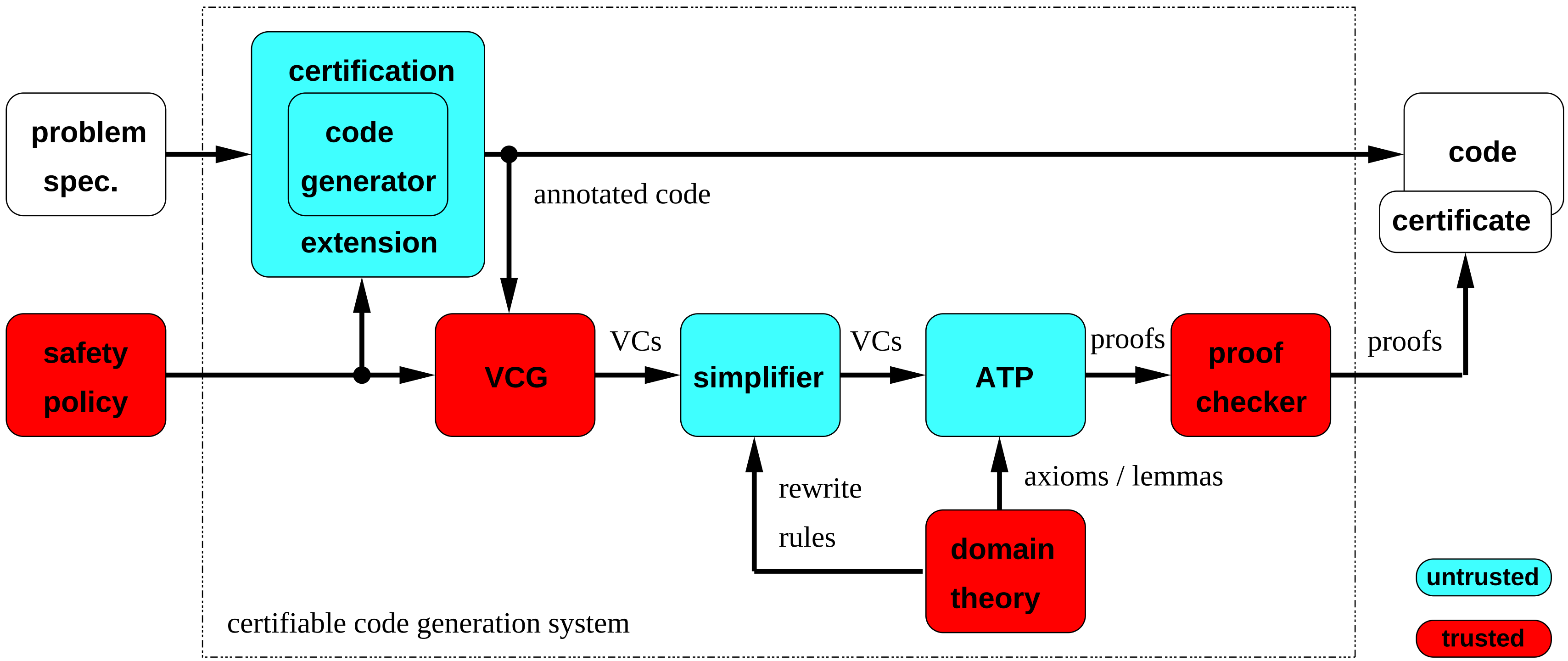
Disdvantages:

- limited — only partial assurance (flip-side of customizable)

Generator Assurance Architectures

Correct-by-construction: “*Trust me, I’m a doctor…*”

Generator Assurance Architectures



Certifiable program generation: *"Don't trust me, I'm a computer scientist…"*

- *Trusted code base* minimized
  - "large" components untrusted
  - trusted components (more) deterministic
- Approach
  - generate safety obligations (i.e., VCG applies safety policy to program)
  - simplify, prove, & check

Outline

1. Introduction (*or: Taking Stock*)
2. Certifiable Program Generation (*or: I have a plan*)
3. **Certification Framework** (**or: Greek Letters**)
4. Annotation Generation (*or: TANSTAAFL*)
5. Experiments (*or: Drosophila and Tables*)
6. Future Work (*or: Wild Speculations*)

Certification Framework

Safety *property*: operational characterization of *intuitively safe* programs

> "*All automatic variables shall have been assigned a value before being used*" (MISRA 9.1)

Formal:

- introduce "shadow variables" to record safety information
- operational semantics (extended by effects on shadow variables):

$$\langle x \;\texttt{:=}\; e, \eta, \bar{\eta}\rangle \Rightarrow \langle \texttt{skip}, \eta \oplus \{x \mapsto [\![e]\!]_\eta\}, \bar{\eta} \oplus \{x_{\text{init}} \mapsto \text{INIT}\}\rangle$$

$$\langle x\texttt{[}e_1\texttt{]} \;\texttt{:=}\; e_2, \eta, \bar{\eta}\rangle \Rightarrow \langle \texttt{skip}, \eta \oplus \{x \mapsto (x \oplus \{[\![e_1]\!]_\eta \mapsto [\![e_2]\!]_\eta\})\}, \bar{\eta} \oplus \{x_{\text{init}} \mapsto (x_{\text{init}} \oplus \{[\![e_1]\!]_\eta \mapsto \text{INIT}\})\}\rangle$$

$\ldots$

Certification Framework

Safety *property*: operational characterization of *intuitively safe* programs

> "*All automatic variables shall have been assigned a value before being used*" (MISRA 9.1)

Formal:

- introduce "shadow variables" to record safety information
- operational semantics (extended by effects on shadow variables)
- semantic safety definition (judgement on expressions and statements):

$$\eta, \bar\eta \models x\ \textsf{safe}_{\text{init}} \quad \text{iff} \quad x_{\text{init}} = \text{INIT}$$

$$\eta, \bar\eta \models x[e]\ \textsf{safe}_{\text{init}} \quad \text{iff} \quad \bar\eta(x_{\text{init}})[\![e]\!]_{\eta,\bar\eta} = \text{INIT} \ \text{ and } \ \eta, \bar\eta \models e\ \textsf{safe}_{\text{init}}$$

$$\ldots$$

$$\eta, \bar\eta \models x[e_1]\ \textsf{:=}\ e_2\ \textsf{safe}_{\text{init}} \quad \text{iff} \quad \eta, \bar\eta \models e_1\ \textsf{safe}_{\text{init}} \text{ and } \eta, \bar\eta \models e_2\ \textsf{safe}_{\text{init}}$$

$$\ldots$$

Certification Framework

Safety *property*: operational characterization of *intuitively safe* programs

"*All automatic variables shall have been assigned a value before being used*" (MISRA 9.1)

Formal:

- introduce "shadow variables" to record safety information
- operational semantics (extended by effects on shadow variables)
- semantic safety definition (judgement on expressions and statements)
- safety reduction (consistency of safety property):

$\eta, \bar{\eta} \models c$ safe and $\langle c, \eta, \bar{\eta} \rangle \Rightarrow \langle c', \eta', \bar{\eta}' \rangle$ implies $\eta', \bar{\eta}' \models c'$ safe

$\Rightarrow$ "safe programs don't go wrong"

Certification Framework

Safety *policy*: proof rules to show that safety property holds for program

- responsible for
  - maintenance of shadow variables
  - construction of safety obligations
- Hoare-rules (extended by safety predicate and shadow variables):

(*assign*) $$\frac{}{Q[e/x, \mathrm{INIT}/x_{\mathrm{init}}] \wedge \mathit{safe}_{\mathrm{init}}(e)\ \{x\ \texttt{:=}\ e\}\ Q}$$

(*update*) $$\frac{}{Q\begin{bmatrix} \mathit{upd}(x, e_1, e_2)/x, \\ \mathit{upd}(x_{\mathrm{init}}, e_1, \mathrm{INIT})/x_{\mathrm{init}} \end{bmatrix} \wedge \mathit{safe}_{\mathrm{init}}(e_1) \wedge \mathit{safe}_{\mathrm{init}}(e_2)\ \{x\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2\}\ Q}$$

(*if*) $$\frac{P \Rightarrow \mathit{safe}_{\mathrm{init}}(b) \quad b \wedge P\ \{c\}\ Q \quad \neg b \wedge P \Rightarrow Q}{P\ \{\texttt{if}\ b\ \texttt{then}\ c\}\ Q}$$

(*while*) $$\frac{P \Rightarrow \mathit{safe}_{\mathrm{init}}(b) \quad b \wedge P\ \{c\}\ P}{P\ \{\texttt{while}\ b\ \texttt{do}\ c\}\ \neg b \wedge P}$$

Certification Framework

Safety *policy*: proof rules to show that safety property holds for program

- responsible for
  - maintenance of shadow variables
  - construction of safety obligations
- Hoare-rules (extended by safety predicate and shadow variables)
- safety predicate $safe_{\text{init}}(e)$ corresponds to semantic safety conditions:

$$
\begin{aligned}
safe_{\text{init}}(x) &\equiv x_{\text{init}} = \text{INIT} \\
safe_{\text{init}}(x[\![e]\!]) &\equiv x_{\text{init}}[\![e]\!] = \text{INIT} \wedge safe_{\text{init}}(e) \\
&\ldots
\end{aligned}
$$

Certification Framework

Safety *policy*: proof rules to show that safety property holds for program

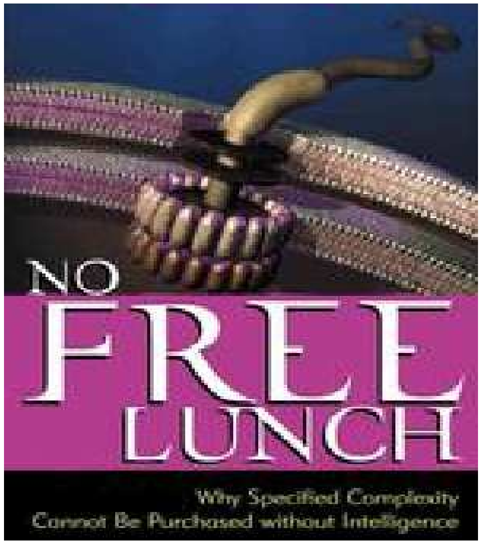
- responsible for
  - maintenance of shadow variables
  - construction of safety obligations
- Hoare-rules (extended by safety predicate and shadow variables)
- safety predicate $safe_{\mathrm{init}}(e)$ corresponds to semantic safety conditions
- soundness and completeness: $\vdash^{\mathtt{safe}} P\ \{C\}\ Q$ iff $\models^{\mathtt{safe}} P\ \{C\}\ Q$

  $\Rightarrow$ off-line proof

Annotation Generation

The Certification Dilemma:

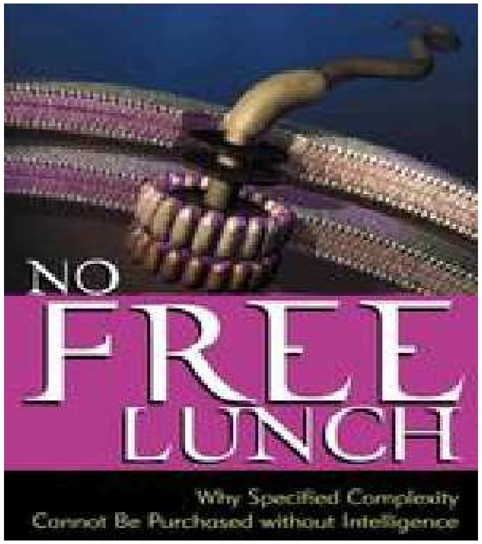
Annotations are crucial but cannot be invented by the machinery.



Annotation Generation

The Certification Dilemma:

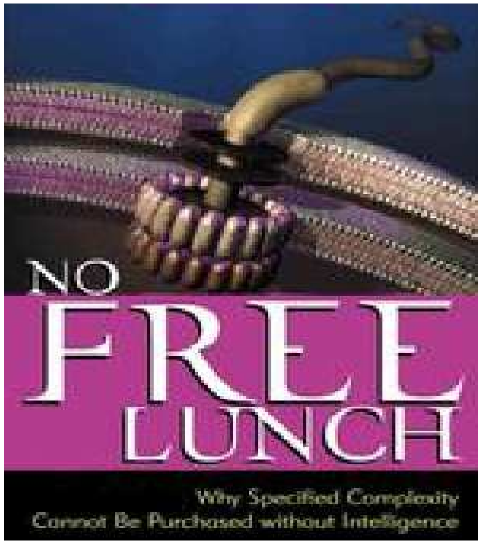
Annotations are crucial but cannot be invented by the machinery and must (ultimately) be provided by the generator developer.



Annotation Generation

The Certification Dilemma:

Annotations are crucial but cannot be invented by the machinery and must (ultimately) be provided by the generator developer.



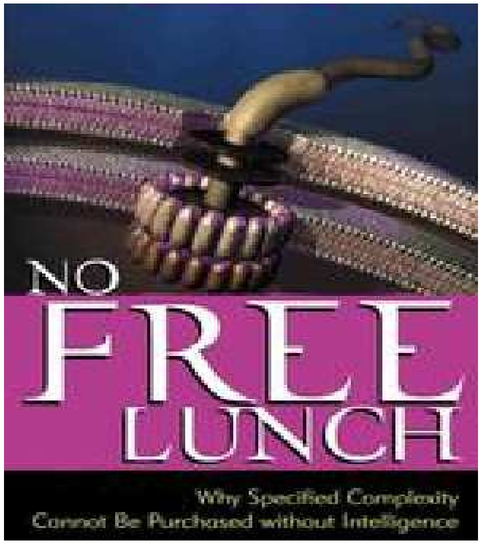
The Bad: It is hard work!

- annotation generation is tedious meta-hack^H^H^H^Hprogramming
- annotations are cross-cutting concerns (object- *and* meta-level)
- annotations are different for each safety property

Annotation Generation

The Certification Dilemma:

Annotations are crucial but cannot be invented by the machinery and must (ultimately) be provided by the generator developer.



The Bad: It is hard work!

- annotation generation is tedious meta-hack^H^H^H^Hprogramming
- annotations are cross-cutting concerns (object- *and* meta-level)
- annotations are different for each safety property

The Good: Everything is known at meta-compile time!

- structure and purpose of generated code limited and known
- safety properties limited and known

Annotation Generation

Example: annotations for *array*-safety:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do      // pick classes randomly

    c[i] := rnd(C);


  ...
  for i := 1 : N do      // set weight for picked class

    for j := 1 : C do w[i,j] := 0.0;
    w[i,c[i]] := 1.0;
  ...
end
```

Annotation Generation

Example: annotations for *array*-safety:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do      // pick classes randomly

    c[i] := rnd(C);


  ...
  for i := 1 : N do      // set weight for picked class
    // inv: 1≤c[i]≤C
    for j := 1 : C do w[i,j] := 0.0;
    w[i,c[i]] := 1.0;
  ...
end
```

Annotation Generation

Example: annotations for *array*-safety:

```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do       // pick classes randomly

    c[i] := rnd(C);
  // post: ∀j· 1≤j≤N ⇒ 1≤c[j]≤C
  ...
  for i := 1 : N do       // set weight for picked class
    // inv: 1≤c[i]≤C
    for j := 1 : C do w[i,j] := 0.0;
    w[i,c[i]] := 1.0;
  ...
end
```

Annotation Generation

Example: annotations for *array*-safety:

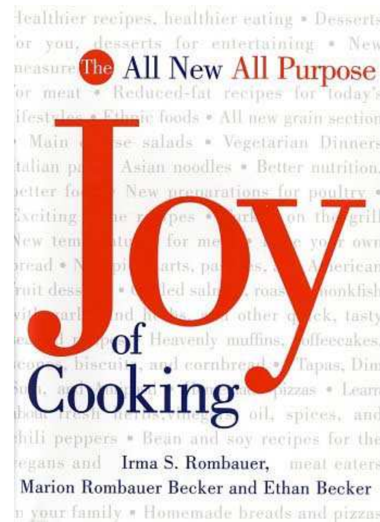
```
begin
  var c[C], w[N,C];
  ...
  for i := 1 : N do      // pick classes randomly
    // inv: ∀j· 1≤j<i ⇒ 1≤c[j]≤C
    c[i] := rnd(C);
  // post: ∀j· 1≤j≤N ⇒ 1≤c[j]≤C
  ...
  for i := 1 : N do      // set weight for picked class
    // inv: 1≤c[i]≤C
    for j := 1 : C do w[i,j] := 0.0;
    w[i,c[i]] := 1.0;
  ...
end
```

Annotation Generation (Meta-level)

Overall recipe:

Repeat until all generated VCs are proven

1. identify structure and location
   of required annotations in code
2. for each annotation, generalize it to meta-annotation
3. for each meta-annotation,
   - write annotation template
   - write meta-program that produces annotation
4. for each location, identify the responsible schema(s)
5. for each schema, integrate meta-annotations



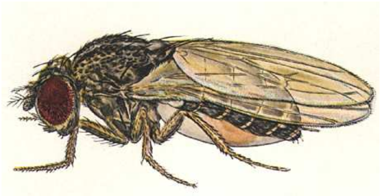
Annotation Generation (Object-level)

At program generation time:

- annotation templates instantiated in parallel with code templates
  - code generator / frontend
- annotations refined in parallel with code
  - code generator / backend
- information propagated “globally” in pre-processing step
  - approximates strongest postcondition transformer

$\Rightarrow$ annotations *not* trusted (i.e., not safety-critical)
  - obligations produced by (trusted) safety policy

Certification Experiments



Drosophila nigrospiracula

Experimental set-up:

- Synthesis systems & test programs:
  - AUTOFILTER: state estimation based on Kalman-filters

    `ds1` – Deep Space 1 attitude estimation

    `iss` – Space Station simulation (part)
  - AUTOBAYES: statistical data analysis

    `segm` – image segmentation via clustering

    `gauss` – image fitting to model
- Safety policies:
  - *array*: $\forall a[i] \in c \cdot a_{lo} \leq i \leq a_{hi}$
  - *init*: $\forall$ *read-var* $x \in c \cdot init(x)$
  - *inuse*: $\forall$ *input-var* $x \in c \cdot use(x)$
  - *symm*: $\forall$ *matrix-var* $m \in c \cdot \forall i, j \,.\, m[i,j] = m[j,i]$
  - *norm*: $\forall$ *vector-var* $v \in c \cdot \Sigma_{i=v_{lo}}^{v_{hi}} v[i] = 1$

Certification Results

Example	$\vert S\vert$	$\vert P\vert$	Policy	$\vert A\vert$	$\vert A^*\vert$	N	N_{fail}	T_{gen}	T_{proof}
`ds1`	48	431	*array*	0	19	1	-	5.5	1
			init	87	444	74	-	11.4	84
			inuse	61	413	21	1	8.1	202
			symm	75	261	865	-	70.8	794
`iss`	97	755	*array*	0	19	4	-	24.7	3
			init	88	458	71	-	39.7	88
			inuse	60	361	1	1	31.6	-
			symm	87	274	480	-	66.2	510
`segm`	17	517	*array*	0	53	1	-	3.0	1
			init	171	1090	121	-	7.6	109
			norm	195	247	14	-	3.6	12
`gauss`	18	1039	*array*	20	505	20	-	21.3	16
			init	118	1615	316	-	54.3	259

Certification Results

Example	$\lvert S \rvert$	$\lvert P \rvert$	Policy	$\lvert A \rvert$	$\lvert A^* \rvert$	N	N_{fail}	T_{gen}	T_{proof}
`ds1`	48	431	*array*	0	19	1	-	5.5	1
			init	87	444	74	-	11.4	84
			inuse	61	413	21	1	8.1	202
			symm	75	261	865	-	70.8	794
`iss`	97	755	*array*	0	19	4	-	24.7	3
			init	88	458	71	-	39.7	88
			inuse	60	361	1	1	31.6	-
			symm	87	274	480	-	66.2	510
`segm`	17	517	*array*	0	53	1	-	3.0	1
			init	171	1090	121	-	7.6	109
			norm	195	247	14	-	3.6	12
`gauss`	18	1039	*array*	20	505	20	-	21.3	16
			init	118	1615	316	-	54.3	259

$\Rightarrow$ formulation of *inuse*-policy too conservative

Certification Results

Real errors caught in generator (anecdotal evidence only...):

- division-by-zero error hidden in schema:
  - generated fragment:

```
for i := 1 : C do          // pick centers randomly
  c[i] := x[rnd(N)];
for i := 1 : N do          // compute weights via distances
  for j := 1 : C do
    w[i,j] := sqrt((c[j]-x[i])**2)
                / sum(k := 1 : C, sqrt((c[k]-x[i])**2));
```

$\Rightarrow$ error manifests itself only if all input data `x[i]` are equal

$\Rightarrow$ caught by *partial-operator*-policy

- uninitialized variable caused by generator maintenance:
  - added simplified version of Kalman-schema (hardcodes $H = 0$)
  - botched "partial evaluation": removed too much code

$\Rightarrow$ caught by *init*-policy right after introduction

Future Directions

- Extend range or safety policies
  - type conformance: units, behavioral subtypes, . . .
  - protocol conformance: locking, separation, . . .
- Support different “reasoning engines”: static analysis
- Apply to other code generators: Simulink/Matlab RealTime Workshop
- Annotation inference
  - seperate code generation and annotation generation
  - infer annotations from code structure and safety policy
  - use AOP-style techniques
  - “go meta-meta”: generate aspects if necessary

$\Rightarrow$ exploits idiomatic structure of generated code

Future Directions

- Extend range or safety policies
  - type conformance: units, behavioral subtypes, ...
  - protocol conformance: locking, separation, ...
- Support different "reasoning engines": static analysis
- Apply to other code generators: Simulink/Matlab RealTime Workshop
- Annotation inference
  - seperate code generation and annotation generation
  - infer annotations from code structure and safety policy
  - use AOP-style techniques
  - "go meta-meta": generate aspects if necessary

  $\Rightarrow$ exploits idiomatic structure of generated code

PCC for code generators!